THREAD CONSISTENCY SUPPORT SYSTEM AND METHOD

COPYRIGHT NOTICE

BACKGROUND OF THE INVENTION

This invention relates generally to software thread management and, more particularly, to a thread connection enhancement to a connector application program interface that provides alternate thread support in the form of thread consistency to connectors.

In computer science terminology, a thread, sometimes shorthand for thread of control, generally represents an execution stream through a software process. Threads can only exist if a process exists. When the process starts, it does all of the required initialization, such as address spacing and the like, which requires use of system resources. Threads allow multiple processes to share the same address space, resources, and so on, while running concurrently. A thread often owns a small or minimal set of resources, including a machine state, a control stack, machine interface (MI) automatic storage, and anchors for thread local storage. Threads share other process resources, such as static and heap storage, program activations, etc., with other threads. A threaded application that makes use of, for example, static storage must synchronize access to that storage among its threads. Neither the programming language, such as C, nor the operating system provides implicit synchronization.

EXPRESS MAIL LABEL NO. EL 920635767US

When an application uses more than one thread at a time, it is generally referred to as multi-threading. If an application is saturated, starting a new thread will often help. Starting a new thread is fast and economical, and it thus improves performance versus starting a new process to do the same job. Unlike processes, one advantage of multi-threading is that there are no resource sharing concerns. While applications vary, multi-threading can provide many benefits, such as improved application scalability due to reduced use of resources, superior response time in creating new threads as compared with new processes, and centralized management of client connections in a single server.

However, some back-end systems, that is, external systems like databases or messaging servers, which provide information access via a published interface, are more restrictive and confine operations to that of a single particular thread per connection, or execution instance which uses the published interface to access the back-end system -- usually the thread to which the back end has granted a connection handle. This leads to incompatibility problems when a multi-threaded client application seeks to interface with this type of restrictive back end system. The restrictive back-end systems may require completely single threaded server applications that maintain thread consistency. In such systems, multiple threads could still exist in a thread consistent environment, provided that each connection is associated with only one thread. In contrast, a multi-threaded server application requires that the connectors support multi-threading, since the connector APIs of these multi-threaded server applications assume that connections may be used freely between threads, and that each connector supports thread execution without regard to thread consistency.

Thus, this type of restrictive back-end system simply will not support multiple threads from a multi-threaded server application. This is unfortunate since many server

EXPRESS MAIL LABEL NO. EL 920635767US

applications are multi-threaded, and it is desirable to be able to implement these applications without regard to the back-end system. Accordingly, there is a need for a system that allows a multi-threaded server application to interface with a restrictive back-end system that requires thread consistency. The present invention fulfills this and other needs.

SUMMARY OF THE INVENTION

The above and other needs are addressed by a method in accordance with the present invention for interfacing between a multi-threaded client application and a restrictive back-end processing system through a thread-dependent connector which generates connections requiring single threads. The method includes detecting a thread-dependent connection request to a back-end processing system, maintaining a single thread to link to the detected thread-dependent connection such as by mapping each of the multiple threads with the single thread, and correlating multiple threads from the application with the maintained single thread. The method funnels operation requests by multiple application threads through the thread-dependent connection's dedicated thread to the back-end processing system. In some embodiments, detecting a thread-dependent connection involves detecting an attempt by the application to access the back-end processing system and then querying the connector to see if it requires single-threaded access to the back end. This may be accomplished by reading a header data structure for the connector which stores data identifying the properties of the given connector including information about its thread dependencies. For example, the header data structure may contain a flag indicating whether the connector supports multiple threads per connection. Alternatively, the thread-dependent connection is detected by detecting a connector allocating memory for an instance of a connection.

In some embodiments, maintaining a single thread involves allocating a thread from the application and isolating it as the single thread to link to the connection handle granted by a back-end system. For back-end systems which support multiple simultaneous connections but still require correlation between each connection and its application thread, the method may further involve detecting a second thread-dependent connection request to the back-end processing system, maintaining a second single thread to link to the detected second thread-dependent connection to the back-end system, and correlating one or more threads from the application with the maintained second single thread. This allows additional operations requested by the application over the second thread(s) to be performed on the back-end processing system through the second thread-dependent connection.

Multiple threads from the application may be correlated by generating a thread support object to support relationships between the multiple threads and the maintained single thread and using the thread support object to toggle execution of an operation between one of the multiple threads and the maintained single thread. The thread support object may contain two or more semaphores which would be used to toggle execution of the operation between one of the multiple threads and the maintained single thread.

The need described above is also addressed by a thread consistency support system of the present invention that produces thread consistency between a multi-threaded application and a thread-dependent connector. The system includes an arbiter or interface layer positioned between the application and the thread-dependent connector. The thread-dependent connector only allows a single thread to link to that connector for all operations on that connector. The multi-threaded application creates multiple threads that attempt to access the thread-dependent connector. The arbiter layer receives requests from the multiple application threads and channels them through a single internal thread associated with the connector, upon which operations of the multiple application threads are performed.

In one embodiment of the present invention, the thread consistency support system further includes an activation detector that activates the arbiter layer in response to the detection of an attempted access to a restrictive back end system which would generate a thread-dependent connector. The activation detector may be an enhancement incorporated into a connector application program interface. In some embodiments the system is configured to channel multiple threads from a multi-threaded application to more than one thread-dependent connector.

In one embodiment of the present invention, the thread consistency support system channels the multiple threads from the application to an isolated, single thread associated with the thread-dependent connector. The thread consistency support system may execute a thread isolation routine to channel the multiple threads from the application into a single internal thread. In one embodiment, the thread consistency support system utilizes threading in connection with a toggle algorithm to isolate thread execution and channel the multiple threads from the application into a single internal thread.

In another aspect of a thread consistency support system constructed in accordance with the present invention, the single internal thread produced by the arbiter layer acts as a thread sub-connection to the thread-dependent connector that is assigned to each of the original multiple thread connections from the application to the arbiter layer. The thread consistency support system may utilize connector methods that appear to the application to be those of the underlying sub-connected thread-dependent connector. In one embodiment, the thread consistency support system isolates a single thread of the sub-connection for use in operations of the back-end system and for interacting with and responding to requests from the multi-threaded application.

In still another embodiment of the present invention, the thread consistency support system includes a threading meta-connector interacting between the connector application program interface and the thread-dependent connector. The threading meta-connector establishes a connection handle for a single internal thread with the thread-dependent connector that is returned to the connector application program interface of a calling application in place of multiple requested thread connection handles. The connection handle is established in response to the threading meta-connector's receipt of a request to allocate a connection to a restrictive back-end system. The same thread that initializes a connection from the thread-dependent connector is then used for some or all subsequent operations attempted by the multiple threads from the application with that connector.

A method for providing thread consistency according to the present invention includes receiving a request to connect to a restrictive back-end system, creating a single thread connection with a thread-dependent connector of the back-end system, and utilizing the single thread connection with the thread-dependent connector for the multiple threads from the connector application program interface to channel operations to be performed by the multiple threads therethrough.

Other features and advantages of the present invention will become apparent from the following detailed description, taken in conjunction with the accompanying drawings, which illustrates by way of example, the features of the present invention.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 illustrates a thread consistency support system of the present invention interfacing with an application and a restrictive back-end processing system;

Fig. 2 illustrates in more detail one embodiment of the thread consistency support system;

Fig. 3 illustrates a multi-threaded application using conventional thread support to interface with a thread-dependent connector; and

Fig. 4 illustrates an exemplary use of the thread consistency support system of the present invention to interface between a multi-threaded application and a thread-dependent connector.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

A thread consistency support system constructed in accordance with embodiments of the present invention provides an interface layer between a multi-threaded application written to a Connector Application Program Interface (API) and an individual connector in a restrictive back-end system. As described further below, the interface layer maps the application's multiple threads to the internal thread(s) of the thread consistency support system in order to preserve thread consistency by, for example, maintaining one thread per connection.

Referring now to the drawings, and more particularly to Fig. 1, a thread consistency support system 10 is provided to interface between a multi-threaded application and a restrictive back-end processing system 8. The multi-threaded application 60 handles processing of its operations through multiple simultaneous processing or software threads 70, as known to those of skill in the art. The back-end processing system 8 is restrictive in that it contains a connector 40 which is thread-dependent and generates instances of connections 42 which each process only a single thread at a time. The thread consistency support system 10 detects the a request made to a restrictive back-end system with the thread dependency and correlates the multiple threads 70 with each single thread 80 for each connection instance 42.

Referring now to an embodiment of the present invention as shown in Fig. 2, a thread consistency support system 10 includes a Detection/Activation routine 20 and a Threading MetaConnector routine 30. The Detection/Activation routine 20 is configured to determine if thread consistency is required. If thread consistency is required, the Detection/Activation routine 20 activates the Threading MetaConnector routine 30 in response to detection of an application attempting to access a restrictive back-end system that uses connectors requiring single threads. The Threading MetaConnector routine 30 is configured to receive multiple incoming application threads 70 from a Connector API 50 and channel the multiple application threads 70 into a single internal thread 80 that is maintained from the thread consistency support system 10 to the connector 40 until the all data operations of the multiple application threads 70 are complete. The Threading MetaConnector routine 30 only allows a single thread to link to a given connector 40 for all operations on that connector.

The Threading MetaConnector routine 30 interfaces between the Connector Application Program Interface 50 and the Thread-dependent Connector 40. The Threading MetaConnector routine 30 isolates a single internal thread for the Thread-dependent Connector. In this manner, thread consistency support system 10 ensures that the same thread that initializes a connection with the Thread-dependent Connector is used for all subsequent operations attempted by the multiple threads 70 from the application 60 with that connector 40. The thread consistency support system 10 of the present invention implements a level of indirection between an application's threads 70 to the one or more connectors 40 and internal threads 80. This involves mapping the application's threads 70 to the internal threads 80 in order to preserve one thread per connection.

In one embodiment, the Connector API 50 is the Lotus Connector API, and the individual connector 40 is the Lotus Domino Connector. The thread consistency support system 10 is only active for connectors which require thread consistency. Connectors requiring support from the thread consistency support system 10 are identified as such through a list of supported threads contained in a header block of the connector 40. When a connector 40 identifies the need for this thread consistency support that is provided by the present invention, the Lotus Connector API automatically activates the thread consistency support system 10 in order to isolate application threads 70 from the internal threads 80 running through the connector 40.

In one embodiment, the Detection/Activation routine 20 may be implemented as an enhancement or extension to a connection allocation function of a Connector API 50, such as the Lotus Connector (LC) Connection Allocation function LCConnectionAlloc() of the Lotus Connector API 50, while the Threading MetaConnector 30 may be entirely new code. The Detection/Activation routine 20 would then be written in the same programming language as the connection allocator function, such as the C programming language. The Threading MetaConnector 30, if implemented as entirely new code, may be written in a different programming language, such as C++ or other languages as known to those of skill in the art.

In a preferred embodiment of the present invention, the thread consistency support system 10 is supported by the following platforms which are listed by way of example only, and not by way of limitation: Windows 2000, Windows NT 4.0, Windows 95, Windows 98, OS/2 Warp 4.0, AIX 4.3.1, Solarius SPARC2.5, HP-UX 11.0, Linux, AS/400, S/390.

Referring now to the Detection/Activation routine 20 of the thread consistency support system 10 (which enhances the abilities of the Connector API 50), in one embodiment of the present invention the Connector API loads a connector as part of the Connection Allocation routine. At load time, the Connector Identity function is called to gather information about the Connector and what support the Connector provides, including whether the connector supports multi-threading. The connector's thread support is part of the identification data in the connector's header block. When queried by the API at load time, the connector provides a list of supported threads. If a requested thread is not in this list, the API interrogates the connector to determine if it will support the thread.

In one embodiment of the present invention, the thread consistency support system 10 uses a flag, called the Identify Context Thread, that is created within the identification structure in the connector's header. This flag indicates whether the connector requires special handling of threads and is a Thread-dependent Connector 40. The flag indicates whether the connector 40 requires that the same thread which initializes the connection must be used for all subsequent operations on that connection. The flag does not indicate the connector is single-threaded, since multiple threads may exist when multiple connections are active at one time. That is, the connections, which are instances of the connector, may be limited to single thread access while the connector is capable of generating additional instances of connections to handle multiple threads.

In an embodiment of the present invention, when a multi-threaded application 60 calls the Connection Allocation routine to load a connector 40, the thread consistency support system 10 tests the connector's thread information. If the Identify Context Thread flag is set, the thread connector code system 10 calls the Connection Allocation routine to load and initialize the

Threading MetaConnector 30. Once the Threading MetaConnector 30 has been created, the original connection is assigned as the Threading MetaConnector's sub-connection. Thus, the connection handle for the Threading MetaConnector 30 is returned to the calling multi-threaded application 60 in place of the requested connection handle. In order to maintain transparency, the Threading MetaConnector 30 reports back the sub-connector's responses (those of the Thread-dependent Connector 40) for all properties including a token connector name. Additionally, the connector's methods appear as those of the underlying sub-connector.

Referring now more specifically to the Threading MetaConnector 30 portion of the present invention, the Threading MetaConnector 30 receives calls from the Connector API 50 and passes those calls on to the Thread-dependent Connector 40. The Connector API 50 may receive calls from multiple threads to a single connection, whereas the Threading MetaConnector 30 creates and uses one thread for the duration of each connection.

In one particular embodiment of the present invention, the Threading MetaConnector 30 allocates a block of memory and creates a context object when a new connection is initialized. This is similar to the function performed generally by connectors and metaconnectors. As part of the context object, the Threading MetaConnector 30 maintains both the handle of the Thread-dependent Connector 40 and a Thread Support Object (TSO). The Thread Support Object consists of two semaphores, a data passing structure, and a thread executing within a message processing routine. In this implementation of the present invention, the thread consistency support system 10 uses threading in connection with a toggle algorithm to isolate thread execution. By utilizing two semaphores, execution toggles between the thread executing from the application and from the Thread Support Object's thread. In the portion of the thread consistency support system 10 that executes the toggling between the application code and the Thread Support Object, a Connector Initialize routine and a Connector Terminate routine are responsible for configuration and cleanup of the Thread Support Object. In one embodiment, remaining connector methods, with the exception of the Connector Identify routine, are processed through the Thread Support Object. Briefly stated, the application code threads 70 run from the multi-threaded application 60 through the Connector API 50 and into the Threading MetaConnector 30, while the Thread Support Object thread runs inside the Thread Support Object processing routine.

The Threading MetaConnector's context block contains elements for the Thread-dependent Connector's connection handle as well as the data passing structure. A pointer to the context block is passed when the Thread Support Object's thread is first created on the processing routine. In one embodiment, the data passing structure contains the union of all parameters which are used as input and output from the methods of the Connector API 50. In this embodiment, various methods are utilized by the Connector API for connectors and meta-connectors. The methods support a variety of connector functions including identifying the connector to the API, performing initialization and termination of a connection, fetching or setting connector property values and data provider records, handling metadata, and performing other desired actions. For example, the Lotus Connector API has a published interface to the connector which supports seventeen methods for any connector or meta-connector. Not all methods are supported by all connectors; however, each is implemented and may return a value of unavailable when not supported.

In one embodiment of the present invention, the Threading MetaConnector 30 has specific code associated with initialization and termination. For all remaining functional methods, the Threading MetaConnector 30 performs parameter loading and unloading, as well as thread toggling. Examples of individual methods for specific embodiments are outlined below in conjunction with details for specific methods, as well as generic method processing.

For example, in the implementation of the present invention in the Lotus Connector API, LCX Identify is a method called when the Threading MetaConnector 30 is loaded. The LCX Identify method returns the MetaConnector's LCX identifying structure. The LCX identifying structure indicates levels of threading support as well as the size of the context block required. The context block contains internal data elements as well as the parameter passing structure for the Thread Support Object for the Lotus Connector methods. Minimal data structures are required for the Thread Support Object parameters in the context block. Additional data elements may also be added as desired.

Another method in this implementation is the LCX Initialize method, which performs internal initialization specific to the current thread, including initialization of the context block. Additionally, if a Process flag is set, this method additionally performs MetaConnector initialization specific to the entire process. Once the context block is initialized, any optional general purpose stream object that exists, which may be part of the context block, is cleared. Additional objects may also be stored in the context block for optimization purposes. Any such objects would also be initialized at this time. As part of the context block, the Thread Support Object is initialized, including the parameter structure and the semaphores necessary for toggling execution of the Thread Support Object processing routine. Moreover, the Thread Support Object thread is spawned on its processing routine. The processing routine initializes any internal data and then waits for the processing request. The context block may be initialized by using a memory set routine to initialize all bytes to zero and then set any non-zero elements to the their initial values.

Another method is a LCX Terminate method, which performs the termination specific to the current thread. If the Process flag is set, this method additionally performs the termination specific to the entire process. Preferably, a Lotus Connector Method ID of the Thread Support Object is set to Terminate ID, and input parameters are loaded into the parameter structure of the context block for the Thread-dependent Connector's LCX Terminate method. In addition to calling the Connection Terminate routine with the Thread-dependent Connector handle, the Thread Support Object performs its internal clean-up and then exits the process routine, thereby terminating the Thread Support Object thread. The Threading MetaConnector 30 cleans up the Thread Support Object including closing the semaphores. If the context block contains additional objects, such as a general purpose stream or allocated buffers, these are freed and cleaned up at this point.

Referring now to Figs. 3 and 4, an illustrative example of the thread consistency support provided by an embodiment of a thread consistency support system 10 of the present invention is demonstrated. In the example depicted in Figs. 3 and 4, a multi-threaded application 60 is shown which has four threads 70. Three of the threads share a common connection and perform a number of data operations. The remaining thread has a separate connection and performs all of its own data operations. Fig. 3 represents a Connector API 50 utilizing conventional thread support. In contrast, Fig. 4 represents the implementation of an embodiment of the present invention with the Connector API 50 utilizing the thread consistency support system 10 in order to maintain thread consistency.

Referring again to the example of Fig. 3, where the Connector API 50 uses conventional thread support, the first thread establishes a new connection, initializes the connection, and then selects and fetches data. A second thread is then created which uses the existing connection to select, fetch, and then update data. A third thread then establishes a new connection, initializes and connects, and then selects and fetches data. Later, the third thread updates the data and then disconnects and terminates the connection. While the third thread is still active, a fourth thread uses the first connection to remove data.

Referring now to Fig. 4, where the Connector API 50 utilizes the thread consistency support system 10 of the present invention, the example begins with the first thread trying to establish a new connection with a restrictive back-end system. The thread consistency support system 10, and in particular the TMC 30, isolates the connection from the requesting thread 70. The TMC 30 then creates its own thread 80 and uses this internal thread to create the new connection. The TMC 30 maintains a correlation between its internal thread 80 and the connection handle. The application's first thread initializes and connects to the connection. Each call from the application 60 to the connection 40 goes through the thread consistency support system 10 which takes the connection handle and uses it to find its own internal thread 80, which is dedicated to the connection 40. The application thread 70 selects and fetches data. Again, the thread consistency support system 10 uses the connection handle to identify its own internal thread 80 and uses the internal thread when interacting with the connection 40.

A second thread 70 is then created by the multi-threaded application 60 which uses the existing connection to select, fetch, and then update data. The TMC 30 identifies its first internal thread 80 as matching the connection handle. A third application thread 70 establishes a new connection. The thread consistency support system 10 creates another internal thread 80 to correspond to the new connection. The application's third thread 70 initializes and connects, and then selects and fetches data through this second internal thread 80. The TMC 30 identifies its second internal thread 80 as matching the connection handle which is then used to complete the data access. A fourth application thread 70 uses the first connection to remove data. The TMC 30, using the connection handle for the first connection, identifies its first internal thread 80 to complete the operation. Later, the application's third thread 70 updates and then disconnects and terminates the connection. The TMC 30 identifies its second internal thread 80 as corresponding to the connection handle and then uses it as necessary. In a more detailed example of the use of the present invention, throughout this process, the individual application threads may perform additional data operations.

The various methodologies described above are provided by way of illustration only and should not be construed to limit the invention. Those skilled in the art will readily recognize various modifications and changes may be made to the present invention without departing from the true spirit and scope of the present invention. Accordingly, it is not intended that the invention be limited, except as by the appended claims.